Target Container: A Target-Centric Parallel Programming Abstraction for Video-based Surveillance

Kirak Hong[†] Bogdan Branzoi[•] Junsuk Shin[†] Stephen Smaldone[•]
Liviu Iftode[•] Umakishore Ramachandran[†]

[†]Georgia Institute of Technology [•]Rutgers University

ABSTRACT

Surveillance systems are some of the most computationally intensive applications. Despite technological advances, low-cost of sensors, and continuous improvement of computer vision algorithms, large-scale reliable surveillance systems have yet to become common. We argue that building effective smart surveillance systems requires a new approach, with a focus on programmability and scalability. While considerable progress has been made in the area of computer vision algorithms, such advances cannot translate to deployment in the large until adequate system abstractions and resource management techniques are in place to ensure their performance.

In this paper, we propose a novel abstraction, the *target container* (TC) that serves as a parallel programming model and execution framework for developing complex applications for tracking multiple targets in a large-scale camera network. The key insight is to allow the domain expert (e.g., a vision researcher) to focus on the algorithmic details of target tracking and let the system deal with providing the computational resources (cameras, networking, and processing) to enable it. The TC corresponds to a target, possibly tracked from multiple cameras. The domain expert provides the code modules for target tracking (such as detectors and trackers) as handlers to the TC system. The handlers are invoked dynamically by the TC system to discover new targets (detector) and to follow existing targets (tracker). The TC system also provides an interface for merging TCs whenever they are determined to be corresponding to the same target.

A TC serves as the resource principal for all the trackers of a particular target, allowing the TC system to fairly schedule the computational resources for their execution. Thus, there is a one-to-one correspondence between a target and a TC that represents the target in the runtime system. This design has the nice property that the system resources needed to sustain target tracking scales up or down proportional to the number of targets being tracked simultaneously as opposed to the number of cameras in the deployment. As a result, the TC system efficiently manages the computational resources leading to an overall scalability of the surveillance system in terms of the number of targets that can be tracked in real time. The TC model also enables a variety of target tracking policies, including target prioritization. Under this policy, the TC system automatically allocates system resources based upon application-specified target priority.

This paper presents the design of a TC system, details of an experimental prototype, and experimental results that confirm the performance benefits of the TC model compared to a conventional thread-based implementation of a surveillance application.

1. INTRODUCTION

Sensors of various modalities and capabilities have been ubiquitously deployed throughout many urban environments. Fear of terrorism, among other criminal activities, have driven cities such as London and New York to employ a broad-based approach to surveillance using closed-circuit video cameras, among other sensor modalities, to monitor the majority of intra-city locations. The overall goal of surveillance is to detect suspicious activities and to track the individuals who perform them.

The conventional approach to surveillance has required direct human involvement, either at the time of video capture or to periodically review archived video recordings. Recent advances in computer vision techniques are now spawning a class of automated surveillance systems, requiring little to no human involvement to monitor targets of interest, detect threatening actions, and raise alerts. To support the goals of surveillance, video images must be processed in real time in order to successfully meet modern security requirements. In large urban environments, processing continuous data from a large number of video sources is a computationally intensive task as computer vision techniques are especially demanding with respect to system resource requirements. As conventional data processing and interpretation tasks migrate from a human-oriented to a systems-oriented model, questions of system scalability and efficient resource management arise.

Automated surveillance systems are typically composed of two core elements: threat detection and target tracking. Detection is typically concerned with monitoring an area to identify predefined threat conditions. Once a threatening activity or condition has been identified, the surveillance system will track any target that is either performing the activity or related to the threat condition. Both detection and tracking occur within the scope of individual video cameras, but may also span across multiple cameras, introducing an inherently parallel/distributed nature to large surveillance systems. The resource requirements for such large-scale deployments potentially scales up along two dimensions: (a) the number of cameras deployed, and (b) the number of targets being tracked simultaneously. Given the resource requirements for even small-scale automated surveillance, designing systems for city-wide surveillance requires high-levels of scalability.

Beyond solving the issues of system resource management

and scalability, modern automated surveillance systems should support the development of large-scale surveillance applications by providing a simple and intuitive programming model. Providing such a model enables domain experts to focus solely on the algorithmic aspects of the application and frees them from the concerns of the complex systems aspects of resource management and task scheduling.

In this paper, we propose a novel abstraction, the *target container* (TC) that serves as a parallel programming model and execution framework for developing complex applications for tracking multiple targets in a large-scale camera network. The key insight is to allow the domain expert (e.g., a vision researcher) to focus on the algorithmic details of target tracking and let the system deal with providing the computational resources (cameras, networking, and processing) to enable it. The TC corresponds to a target, possibly tracked from multiple cameras. The domain expert provides the code modules for target tracking (such as detectors and trackers) as handlers to the TC system. The handlers are invoked dynamically by the TC system to discover new targets (detector) and to follow existing targets (tracker). The TC system also provides an interface for merging TCs whenever they are determined to be corresponding to the same target.

A TC serves as the resource principal for all the trackers of a particular target, allowing the TC system to fairly schedule the computational resources for their execution. Thus, there is a one-to-one correspondence between a target and a TC that represents the target in the runtime system. This design has the nice property that the system resources needed to sustain target tracking scales up or down proportional to the number of targets being tracked simultaneously as opposed to the number of cameras in the deployment. As a result, the TC system efficiently manages the computational resources leading to an overall scalability of the surveillance system in terms of the number of targets that can be tracked in real time. The TC model also enables a variety of target tracking policies, including target prioritization. Under this policy, the TC system automatically allocates system resources based upon application-specified target priority.

This paper makes the following four contributions:

- It introduces the *target container* (TC) abstraction, which allows a per-target scheduling of system resources for target tracking, while also providing a simple and intuitive interface for developing complex surveillance applications.
- It describes the TC parallel programming model and framework, which simplifies the development of surveillance systems, freeing domain experts from the complex systems concerns of synchronization, scheduling, and resource management.
- It presents the TC execution model, which improves surveillance system scalability through resource management and target prioritization.
- It presents the design of a TC system, details of an experimental prototype, and experimental results that confirm the performance benefits of the TC model compared to a conventional thread-based implementation.

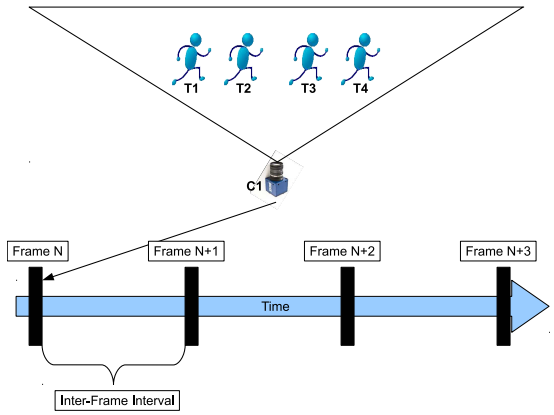
Section 2 further motivates the core problem addressed by the TC system. Section 3 describes the programming model and interface presented to application developers by the TC system along with the details of the TC system design and architecture. Section 4 describes the TC prototype implementation and Section 5 presents the results of our prototype evaluation. Finally, Section 6 positions TC within the broader context of related work in the area and Section 7 concludes the paper.

2. MOTIVATION

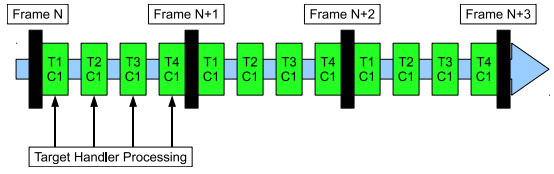
Let us first understand the limitations of the thread-based parallel programming model in developing a large-scale video-based surveillance system both from the point of programmability and resource management. The conventional approach to building surveillance systems is to have the application developer handle all aspects of the system, including traditional systems aspects, such as resource management, and more application-specific aspects, such as mapping targets to cameras. Under this model, a developer wishing to exploit the natural concurrency of the problem has to manage the concurrent programming tasks as part of the application logic. For example, the automated surveillance system should be able to autonomously track a target across the fields of view of multiple cameras, in parallel. Assuming modern computer vision algorithms are able to detect a threat and track a target within a single camera view, the main task of the multi-camera tracking application is to compare different views from different cameras and coalesce them if they represent the same target. This application model implies a number of programming challenges including synchronization and scheduling.

First, to achieve the most efficient parallel implementation, careful synchronization is required for all shared data structures. In the context of automated surveillance, the sharing of target data structures between the detector and tracker application components poses several challenges that must be overcome by the developer. Each detector (one per camera), finds new targets and hands off the detected targets to the trackers (one per target), for continuous monitoring. Target data structures are shared by these components to allow detectors to ensure target uniqueness, and to provide updated information for trackers to continuously monitor their targets. Multiple trackers operating on different video streams may also need to share target data structures when they are monitoring the same target. These complex patterns of data communication and synchronization place an unnecessary burden on an application developer, which is exacerbated by the need to scale the system to hundreds or even thousands of cameras and targets in an urban setting.

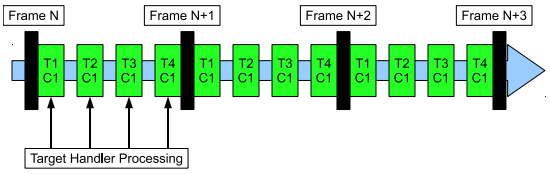
Second, trackers must execute their target monitoring tasks within the resource constraints of the computational infrastructure. A developer is faced with either using the default resource management provided by the computational infrastructure (e.g., pthreads runtime), which is generic and not target-aware, or the developer must handle resource management directly at the application level – a daunting responsibility to say the least. Consider a typical multi-camera multi-target tracking system. Such a system has several avenues for exploiting parallelism. However, there could be data dependencies among the trackers due to the sharing of target data structures, i.e., when multiple trackers are following the same target. Such sharing could lead to inefficiencies due to the contention for shared data structures when trackers following the same target are scheduled to execute concurrently. Ideally, trackers associated with distinct targets should be scheduled concurrently to avoid such



(a) Single Camera Scenario Setup



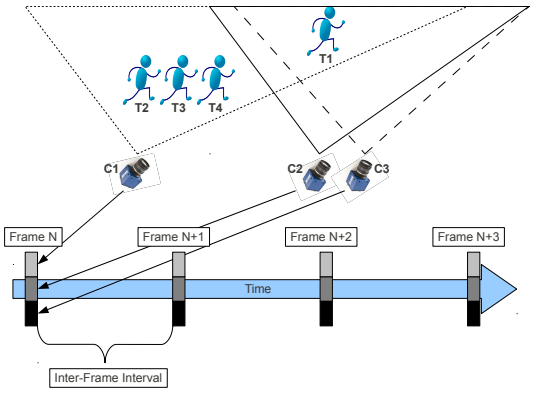
(b) Target Processing (4 Targets / 1 Camera)



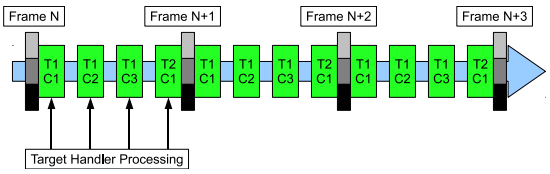
(c) Target Processing (4 Targets / 1 Camera) - TC

(a) For the single camera scenario, video frames arrive at fixed intervals, which contains four targets (T1-T4) in the field-of-view (FOV) of one camera (C1). The time between two consecutive frames is when target handler processing occurs for each target; (b) and (c) show the target handler processing occurring for the conventional and TC cases. Each green bar represents the handler execution for a target in the FOV of a specific camera. For both cases, all target handlers complete within the inter-frame interval.

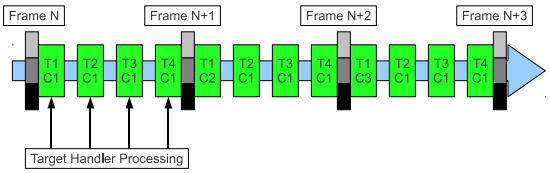
Figure 1: Single Camera Scenario



(a) Multiple Camera Scenario Setup



(b) Target Processing (4 Targets / 3 Cameras)



(c) Target Processing (4 Targets / 3 Cameras) - TC

(a) For the multi-camera scenario, one frame per camera arrives every time interval (shown as a stacked bar). Targets T1-T4 are in the field-of-view (FOV) of camera C1, while T1 is also in the FOV of C2 and C3; (b) and (c) show target handler processing for the four targets in the conventional and TC cases, respectively. Each green bar represents the handler execution for a target in the FOV of a specific camera. Since T1's target handler is executed first, only two targets (T1 and T2) can be updated within the inter-frame interval for the conventional case as shown in (b). Since TC executes target handlers in a target-aware fashion, the handler completion throughput is increased from 2 to 3 as shown in (c). Note that in the TC case, only one camera frame is processed for T1 in each interval leading to an incomplete update of the target.

Figure 2: Multiple Camera Scenario

synchronization overheads. Further, the scheduler should also consider architectural properties such as cache affinity in scheduling the trackers for improved performance.

To better illustrate the scalability challenge in the presence of resource oversubscription, we present two example video surveillance scenarios. In each scenario, we consider one or more cameras to be monitoring a potentially overlapping field of view (FOV). Within the FOV of each camera, there may be one or more targets detected and being tracked by the surveillance system. Data samples from the cameras, called *frames*, arrive at fixed intervals determined by the capture rate of each camera (in frames per second or FPS). A *target handler* is the surveillance application-specific code that is executed to track a target within a single camera view. Whatever processing a target handler performs, to operate in real time, the handler must be executed completely prior to the arrival of the next video frame in sequence.

For simplicity's sake, we consider only the CPU resource. The first scenario (Figure 1(a)) represents no oversubscription, while there is oversubscription in the second scenario (Figure 2(a)). Both scenarios represent a situation wherein the system is monitoring four targets. In the first scenario (Figure 1(a)), all four targets are in the field of view (FOV) of a single camera, while in the second scenario (Figure 2(a)), targets T2-T4 are in the FOV of one camera, while T1 is in the FOV of three cameras. The tracking work to be done for each of the targets in the first scenario is roughly the same (process one video frame in every inter-frame arrival interval). On the other hand, the tracking work to be done for target T1 is three times more in the second scenario since it is in the FOV of three cameras. For the sake of exposition, let us assume that the CPU is capable processing one frame for each of the four targets in the inter-frame interval. Further, let us assume that all four targets are of equal priority. We will postulate that the goal of the system is to achieve the maximum throughput, i.e., number of target updates per inter-frame interval. We do not consider partial completion of the tracking work to be a successful one from the point of view of target update.

The CPU resource is not oversubscribed in the first scenario with either the conventional thread-based approach (Figure 1(b)) or with the TC approach (Figure 1(c)). The

tracking work for all four targets can be completed in the inter-frame interval and the throughput of the system is four target updates in each interval. Conversely, in the second scenario (Figure 2(a)), the CPU is oversubscribed. It has to process six video frames (one each for targets T2-T4 and three for target T1) in the inter-frame arrival interval. The conventional approach successfully completes the tracking work for two targets (T1 and T2) as shown in Figure 2(b). One could think of other interleavings of the processing of the video frames with the conventional approach, but the bottom line is the target throughput in an inter-frame interval in most such interleavings would be close to two. This is because the conventional approach is unaware of targets and does not allocate resources based on targets. On the other hand, Figure 2(c) shows the schedule with the TC approach. Since it makes resource allocation decisions based on targets, it deterministically processes one video frame for each target in the inter-frame interval. Thus, it successfully completes the target updates for three targets (T2-T4) in each inter-frame interval yielding a target update throughput of three. In other words, TC sacrifices accuracy for tracking a target that requires an unfair share of the available resources in favor of completing as many target updates as possible.

Although this example has illustrated how an TC-based target-aware scheduling scheme can substantially improve target update throughput, choosing this scheduling scheme does not preclude more complex schemes, including prioritization. The TC system can utilize target-awareness to not only improve overall throughput, but to provide guarantees to specific high priority targets (see Section 5).

3. ARCHITECTURE OF THE TC SYSTEM

The TC system caters to two complementary goals:

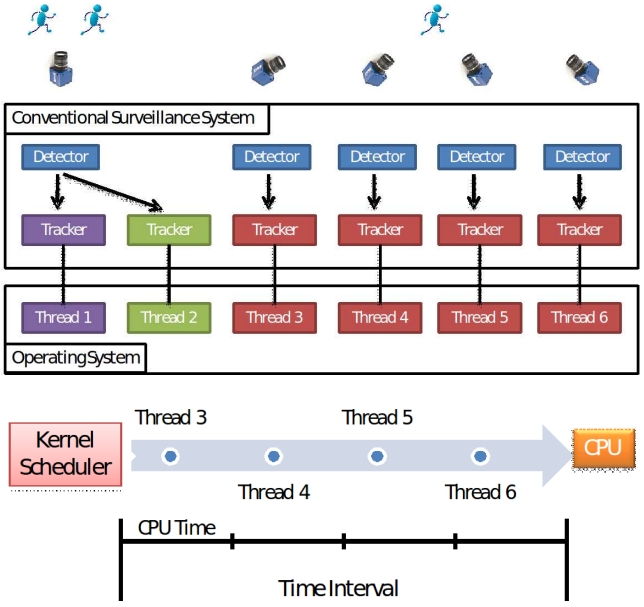
Intuitive programming model. Video-based surveillance systems are inherently parallel and distributed. Consequently, there is an undue burden on the application programmer to deal with the thorny issues of parallel programming including synchronization and scheduling, which affect both the correctness and performance of large-scale surveillance systems. Therefore, the first goal is to reduce the burden on the domain expert by providing an intuitive programming model that subsumes such thorny issues.

Execution model to enforce fair resource management. An equally important goal is efficient use of available computational resources (CPU cycles, memory, and network bandwidth). The goal is to dovetail the programming model with an execution model that enforces fairness in the allocation of resources to ensure scalability of metrics that are meaningful at the application level. There are two such metrics in the context of video-based surveillance applications: *target throughput* and *target latency*. The former is defined as the number of targets successfully updated per unit time. The latter is the time needed on an average to process a target update.

We will elaborate how the TC system accomplishes these goals in the rest of this section.

3.1 Conventional Model

A typical surveillance system is comprised of multiple distributed cameras. The system is capable of running various target detection algorithms for each camera stream. For the purposes of this paper, we refer to the handler running the detection algorithm as the *detector*. The *detector*'s job is to



In this scenario, three targets create six trackers in a conventional system. The kernel scheduler gives each tracker a fair share of the CPU time. The scheduler picks trackers 6, 5, 4, and 3, all following target 3. Target 1 and 2 never get a chance in the chosen time interval.

Figure 3: Target Updates in Conventional Model

recognize new targets in a frame. Once detected, a target needs to be tracked by a tracking algorithm. The handler that performs the tracking within a single camera's FOV is called a *tracker*. The *tracker* is thus associated with a camera and it constantly updates the target data. This data is being used by the detector to detect new targets. To recognize a physical target observed by multiple cameras, an application has to compare different observations and associate those trackers following the same target.

This model implies different levels of parallelism for target tracking. The simplest design choice to exploit parallelism will be a thread per tracker: as a target moves, multiple trackers would be dispatched as independent threads by the detectors associated with the different cameras. Since the trackers process different frame data, their execution is mostly data-parallel even though they constantly update shared target data. However, this approach cannot achieve target-aware resource management. Typically, CPU schedulers in most modern operating systems deal with threads as the unit of scheduling, and assign resources to threads in a fair manner. Thus, a target that is tracked by multiple threads will have an unfair advantage over a target tracked by fewer threads in terms of resource allocation. Also per target priority control is tricky since a typical system provides priority control on a per thread basis. To better illustrate this concept, consider the example in Figure 3.

In the example, three targets are detected by the surveillance system. Target 3 is tracked by four cameras while Targets 1 and 2 are tracked by only one camera. If we assume, in this scenario, that only four trackers can complete their execution within a given time interval, chances are high that either Target 1 or 2, or both, will not get updated. In other words, target throughput, a key metric in surveillance applications, will take a hit due to the unfairness in resource allocation based on threads. Further, if either target 1 or 2

(or both) is of high importance, the prospect of either one not getting proper attention is unacceptable.

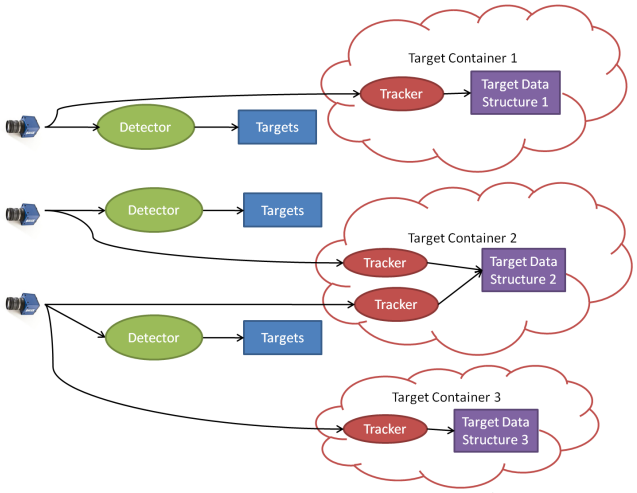
An alternative design would be to have one thread per target. Assuming the underlying system provides fair resource allocation per thread, physical targets with the same priority will receive the same amount of resources. However, this design prevents the use of available fine-grained parallelism, i.e., parallel execution of trackers. Moreover, a typical system provides only limited control of thread priorities, which makes various application-level per-target priority control challenging, if not impossible.

The main problem with these two models is that the underlying system is not aware of targets. A target is an application-level entity and therefore invisible to the operating system from the point of resource management. To implement target aware resource management and scheduling requires a careful design of an appropriate abstraction, which takes into account both the application-level and system-level entities. In this paper, we describe such a design through the TC system, which provides an intuitive programming model and execution framework that liberates the programmers from having to worry about such issues that are mundane but crucially important to achieve good performance for a large-scale automated surveillance system.

3.2 TC Programming Model and API

The intuition behind the TC programming model is quite simple and straightforward. Figure 4 shows the conceptual picture of how a surveillance application will be structured using the new programming model. In principle, the model generalizes to dealing with heterogeneous sensors (cameras, RFID readers, microphones, etc.). However, for the sake of clarity of the exposition, we adhere to cameras as the only sensors in this paper. The application is written as a collection of handlers. There is a *detector* handler associated with each camera stream. The role of the detector handler is to analyze each camera image it receives to detect any new target that is not already known to the surveillance system. The detector creates a *target container* for each new target it identifies in a camera frame. In the simple case, where a target is observed in only one camera, the target container contains a single *tracker* handler, which receives images from the camera and updates the target data structure in every time interval. The time interval is typically the inter-arrival time between two camera frames. However, due to overlapping fields of view, a target may appear in multiple cameras. Thus, in the general case, a target container may contain as many trackers as the number of camera streams, which need to be executed to update the target data structure in every time interval.

In addition to the detectors (one for each sensor stream), and the trackers (one per target per sensor stream associated with this target), the application must provide additional handlers to the TC system for the purposes of merging TCs as explained below. Upon detecting a new target in its field of view, a detector would create a new target container. However, it is possible that this is not a new target but simply an already identified target that happened to move into the field of view of this camera. To address this, the application would also provide a handler for *equality checking* of two targets. Upon establishing the equality of two targets, the associated containers will be merged to encompass the two trackers (see Target Container 2 in Figure 4). The application would provide a *merger* handler to accomplish this merging of two targets into a single one. Incidentally, the application may also choose to merge two distinct targets into a single one (for example, consider a potential threat situation when two cohorts join together and walk in unison in an airport).



A surveillance application is structured as a collection of independent handlers. A detector handler is associated with each sensor instance; its execution results in identifying a set of potential "targets". There is one instance of a tracker handler per target per sensor. Execution of a tracker results in updating the associated target data structure for that target. The target container encompasses all the trackers that need to be run for a successful update of the target data structure associated with that target.

Figure 4: Surveillance Application using TC Model

The intention in this high-level programming model is to provide minimal interface to shield the application developer from common complexities, such as synchronization and scheduling, allowing her to focus on the algorithmic aspect of target tracking. Table 1 summarizes the APIs provided by the TC system for constructing complex surveillance applications. *TC_register_handlers* allows the main logic of an application to provide a number of handlers to the TC system. Once this function is called, the handlers are registered with the TC system and are called upon the occurrence of certain events such as frame arrival or target detection. *TC_create_target* will create a new TC to encompass all data and computation for tracking a target. A tracker can call *TC_stop_track* to notify the TC system that this tracker need not be scheduled anymore; it would do that upon realizing that the target it is tracking is leaving the camera's FOV. *TC_update_data* and *TC_read_data* are used to read and update application-specific target data.

When programming a target tracking application, the developer has to be aware of the fact that the handlers may be executed concurrently. Therefore, the handlers should be written as sequential codes with no side effects to shared data structures to avoid explicit application-level synchronization. However, trackers may have to update target data structures that are potentially shared among several trackers in a target container. For this purpose, the system provides an API that allows reading and updating shared data structures in a mutually exclusive manner. Basically, these API calls subsume data access with synchronization guarantees similar to what has been done by other researchers [2, 17, 18].

API	Description
TC_register_handlers()	Registers all the application handlers with the TC system. This would include the detector, the tracker, the equality checker, and a merger. The detector implements the algorithm for detecting targets in the field of view of the camera, identifying new targets, and creating TCs for new targets. The tracker implements the target tracking algorithm. The equality checker implements the application logic for checking the equality of two targets. Finally, the merger implements the application logic for merging two TCs.
TC_create_target()	It creates a TC and associates it with the new target. This is called by a detector. It also associates a tracker within this TC for this new target that analyzes the same camera stream as the detector.
TC_stop_track()	When a target disappears from a camera's FOV, tracker makes this interface call to prevent further execution of itself.
TC_update_data()	This will be used by detector/tracker for updates to shared data structures.
TC_read_data()	This will be used by detector/tracker for read access to shared data structures.
TC_change_priority()	This will be used by a tracker to change the priority of the TC it is associated with.

Table 1: Target Container API

```
void Detector()
{
  List<BLOB> new_blob_list;
  List<BLOB> old_blob_list;
  List<TC> tc_list;

  while(1)
  {
    IMAGE img = query_stream();
    new_blob_list = detect_blobs(img);

    for_each(tc in tc_list)
    {
      TCData data = TC_read_data(tc);
      old_blob_list.add( data.blob );
    }

    for_each(nB in new_blob_list)
    {
      bool is_new = true;

      for_each(oB in old_blob_list
        If( blob_overlap(nB, oB) == TRUE)
          is_new = false;

      if(is_new == TRUE)
        TC_create_target(nB, img);
    }
  }
}
```

```
void Tracker(TC tc, CAM cam, IMAGE img)
{
  TCData data = TC_read_data(tc);

  BLOB old_blob = data.blob;

  BLOB new_blob = color_track(img, old_blob);

  If( is_out_of_FOV( new_blob ) )
    TC_stop_track(tc, cam);

  data.blob = new_blob;
  data.hist = calc_hist(img, new_blob);

  TC_update_data(tc, data);
}


bool Equality_checker(TC tc1, TC tc2)
{
  TCData data1 = TC_read_data(tc1);
  TCData data2 = TC_read_data(tc2);

  HISTOGRAM hist1 = data1.hist;
  HISTOGRAM hist2 = data2.hist;

  If( compare_hist( hist1, hist2) > THRES )
    return TRUE;

  return FALSE;
}
```

Figure 5: Example Application Code using TC API

A pseudocode example of a multi-camera target tracking application using the TC API is presented in Figure 5. There are three handlers implemented in the code: *detector*, *tracker*, and *equality checker*. Note that *merger* is not shown since its implementation solely depends on application-specific data structures, without the use of any TC function. As shown in the code, the detector discovers a new object within a camera's FOV by comparing the new-blob-list(obtained from the blob entrance detection algorithm) and the old-blob-list(from existing targets). Note that old-blob-list is periodically updated by the target trackers. If the detector finds a new blob that does not overlap with any other existing targets, it creates a new TC by calling *TC_create_target*.

Once created, a tracker tracks a target within a single camera's FOV using a computer vision algorithm. In this example, the algorithm is assumed to be implemented in *color_track* function. The color_track function computes the position of the target in the image (new-blob in Figure 5); using this the tracker updates the new position and the color histogram of the target. If the target is no longer in the image as determined by the color_track function (i.e., the target has left the FOV of the camera), the tracker requests the system to stop scheduling itself by calling *TC_stop_track*. An equality checker compares two color histograms and returns TRUE if the similarity metric exceeds a certain threshold. Note that Figure 5 is an overly simplified illustration of the application logic for demonstrating the use of the TC system API. A sophisticated application may contain much more information than a blob position and a color histogram to represent the target data structure. For example, the target data structure may contain a set of color histograms for different camera views of the same target. The merger handler (not shown in the figure) will be correspondingly more sophisticated, effectively merging two application-specific data structures into one.

3.3 TC Execution Model

The programming model presented in the previous subsection naturally leads to an execution framework that allows metering the resource consumption of a target container. As we mentioned, each TC is associated with a target. In principle, the unit of resource consumption (e.g., CPU scheduling) is a TC independent of the number of trackers contained in it. This ensures that the resource allocation policy is fair across all targets and not biased towards TCs that require the execution of many trackers per target update. Of course, when a target is in the field of view of multiple cameras, at least initially, there will be multiple TCs created since a detector is incapable of knowing the equivalence of a new target it creates with respect to other existing targets created by other detectors. This situation would lead to a higher resource consumption for this target until such duplicate TCs are merged into a single TC containing multiple trackers.

The execution model guarantees target fairness independent of the amount of work that needs to be done per target update. The TC scheduler allocates a time quantum proportional to the expected execution time of a tracker handler. While there can be variance in the actual running time of the tracker depending on the input, as a first order of approximation, we assume that the tracker running time is the same independent of the target. The execution model uses a round robin schedule of TCs. If a TC has multiple trackers, then it will require multiple passes of the round robin scheduler to complete its execution compared to a TC that has a single tracker. This execution model ensures better target throughput and lower average target latency, the two metrics of relevance for surveillance systems. The execution model should be reminiscent of shortest job first (SJF) non-preemptive CPU scheduler. Similar to SJF, there is potential for starvation of TCs with multiple trackers.

The TC execution model also enables a variety of target

tracking policies, including target prioritization. Under the prioritization policy, the TC system automatically allocates system resources based upon programmer-specified target priority, enabling a TC-based surveillance system to accurately track high-priority targets even under conditions of system saturation (i.e., when many targets are being tracked). The priority of a TC can be changed dynamically by the application using the API call *TC_change_priority* (see Table 1). Such changes would be warranted when a tracker observes that a target it is tracking is gaining in importance or vice versa.

3.4 TC Merge Model

As we mentioned earlier, an application developer supplies two functions to the TC system: an *an equality checker* and a *merger*. An equality checker is a binary operator which takes two application specific target data structures from different TCs as arguments. If the two target data structures represent the same target, it returns TRUE, otherwise FALSE. When the *equality checker* returns TRUE, the TC system is ready to combine the two TCs into one. It calls the merger function to merge the application specific data structures, gets rid of one TC, and stores all the necessary metadata pertinent to the merged targets in the other remaining TC.

To seamlessly merge two TCs into one while tracking the targets in real time, the TC system periodically calls equality checker on candidates for merge operation. When merging, the TC system ensures that none of the trackers in the two TCs are running to prevent any side effects. After merge, one of the two TC is eliminated, while the other TC becomes the union of the two previous TCs.

Execution of the equality checker on different pairs of TCs can be done in parallel since it does not update any target data. Similarly, merger operations can go on in parallel so long as the TCs involved in the parallel merges are all distinct.

On the one hand, the merger operation can be viewed as an optimization since it leads to fair allocation of resources to targets. This is especially important under conditions of system overload when the computational resources are oversubscribed. Target update throughput will be adversely affected due to this oversubscription; merging TCs that are tracking the same target will reduce the oversubscription and help in improving the target update throughput. However, the merger operation itself requires computational resources. The ensuing dilemma as to how frequently merger operations should be performed is akin to the thrashing phenomenon observed in virtual memory systems. The paging daemon may have to be run more frequently when the page fault rate goes up to reduce thrashing; similarly, the merger operation may have to be run more frequently under conditions of oversubscription of computational resources (i.e., when the observed target throughput rate goes below a threshold).

On the other hand, merger operation is also a key to ensuring accurate situation awareness. For instance, if two TCs are tracking the same target, this could lead to erroneous inference at the application level. Therefore, from the point of view of ensuring correctness at the application level, it may be necessary to run the merger operations with a certain periodicity. The periodicity should itself be tunable commensurate with the dynamics of the application (e.g., how fast a target is moving).

The TC system can use static information such as camera proximity for selecting merge candidates. If two cameras are geographically too far away, the TCs associated with them are unlikely candidates for a merge operation. There are a number of interesting issues with respect to the sophistication of the equality checker, the logic for selection of merge candidates, the frequency of running the merge operations, and the effects of such policies on target update throughput and target latencies. Such issues are outside the scope of this paper and are part of our future research.

4. IMPLEMENTATION

The TC system architecture is general and lends itself to efficient implementation on parallel and/or distributed platforms. As a proof of concept, we presently have an SMP implementation of the TC system architecture. The state-of-the-art for structured data sharing and synchronization in a distributed system is quite mature [2, 17, 18, 11, 12]. Therefore, extending our current implementation to a distributed system is straightforward.

4.1 Implementation Details

Our current implementation on an SMP is in C++ and uses the pthreads library. The TC runtime system implements each detector handler as a dedicated pthread associated to the specific camera. We maintain a pool of worker threads in the runtime system for scheduling the execution of the tracker handlers. In keeping with the design principle outlined in the system architecture to ensure target fairness, the runtime system executes the trackers in a round-robin fashion across all of the TCs using this pool of threads.

The direct benefit of the user-level design of the TC system using a thread-pool is a reduction in the context switch overhead, since a worker thread simply looks for trackers to run in the TC's run-queue, without needing mediation from the kernel. The number of worker thread in the thread-pool of the TC system is commensurate with the number of CPUs available in the SMP; therefore, this design will lead to much less operating system overhead compared to a conventional implementation using a thread per tracker (see Section 3.1). Further, being above the operating system, the TC system is independent of the scheduling policy of the underlying operating system.

The TC scheduler uses a two-level round robin scheduling policy for TCs and their trackers. First it schedules TCs in a conventional round robin manner. Each time a TC is scheduled, a tracker within the TC will be scheduled. The trackers within a TC are also scheduled in round robin manner so that the next tracker will be executed when next time this TC is scheduled. Each tracker is run to completion in a non-preemptive manner. If a TC has multiple trackers, then it will require multiple time slices to complete its execution compared to a TC that has a single tracker.

To illustrate the execution model, consider the scenario in Figure 6. Here four targets are mapped to four TCs where TC3 and TC4 have two trackers and TC1 and TC2 have only one tracker. Frames from cameras are arriving at the point marked with dashed lines, and therefore there are two frame arrivals in this example. While there can be variance in the actual running time of the tracker depending on the input, for simplicity, we assume that the tracker running time is the same independent of the target. Since there are

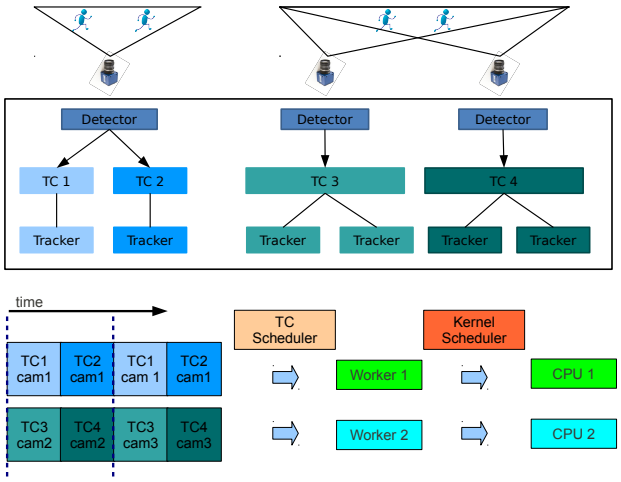


Figure 6: Target Updates in TC Execution Model

two worker threads, the TC scheduler assigns two trackers to the two worker threads in each time slice. As shown in the figure, TC3 and TC4 need two time slices to complete their respective target updates; while TC1 and TC2 are able to complete their target updates in each time slice. In other words, all TCs get a fair share of the available resources.

4.2 Exploiting Cache Affinity in the Scheduling

The TC system uses cache-affinity based scheduling to further optimize system performance. When different TCs share the same frame from a camera, the scheduler executes the trackers in a way that minimizes initial cache misses. For example, in Figure 6, TC1 and TC2 are assigned to the same worker thread, so that the TC2's tracker execution does not have initial cache miss overhead for loading a frame. Also, the scheduling order for TC3 and TC4 ensures that trackers sharing the same frame run with no intervention. Since each frame size for computer vision processing is often large and target tracking is mostly compute-intensive once frame is loaded, this optimization would reduce the number of cache misses significantly and therefore improve performance. However, the TC system uses this optimization only when it does not violate target-level fairness and prioritization.

4.3 Target Priorities

Our prototype also implements application-specifiable target priorities. When a new target is identified by a detector, an initial priority is assigned to the target (i.e., the TC associated with this target) as specified by the application. Priorities range from 1 to 10, allowing an application to assign priorities along a continuous scale. The round-robin TC scheduler respects this priority in assigning a worker thread from the thread pool to execute a tracker. When a worker in a thread pool is available, the TC with the highest nice value will be scheduled. Other TCs will increase their nice value depending on their priorities. If a TC has a higher priority, then it will increase its nice value more than the others, which increases its chance to be scheduled in the next time slice.

4.4 Application Logic

In our current implementation, each *detector* runs the *Foreground Detection* and *Blob Entrance Detection* algorithm implementations from the open source OpenCV library [1]. A *tracker* is an implementation of the color-based tracking algorithm provided by an application. The main task of a tracker is two-fold: *(i)* tracking an object identified by a detector and *(ii)* generating tracking data as a result. While tracking, it updates an object appearance model based on the color histogram of the object. The tracking algorithm from the OpenCV library internally utilizes a technique called mean shift [6], in which the color histogram uses 180 hue levels, and the appearance model uses 256 saturation levels.

To merge different TCs into one when they are tracking the same physical target, the runtime system uses the application provided handlers for equality checking and merger operation. A *merger* handler would aggregate the target feature data structures from multiple TCs (i.e., trackers in those TCs) and return a combined target feature data structure to the runtime system. For example, in our prototype implementation of the surveillance system, we generate an average color histogram model from multiple trackers. An *equality checker* is another handler provided by the application, which decides whether two TCs are following the same target or not. In our prototype, it compares color histograms using Bhattacharyya distance [5] between two histograms.

To verify the advantages of the TC system in terms of both programming and execution models, we have implemented a target tracking application using the TC system. The original code base for the application is from the OpenCV library. The application detects and tracks a number of targets within video streams that are provided by the user. It is important to note that the accuracy of target tracking can vary dramatically depending on the choice of computer vision tracking algorithms and real world conditions such as illumination. However, our focus is not on the validation of the detection or tracking algorithms. Our goals in constructing the example vision application are two-fold: (1) to serve as a proof-of-concept to understand the utility of the TC programming model for rapidly prototyping a surveillance system from off-the-shelf computer vision components; and (2) to serve as vehicle for experimentation and validation of the claims we have made regarding the efficiency of the TC execution model compared to the conventional one.

5. EVALUATION

The goal of this evaluation is to evaluate the scalability of the TC system compared to a conventional video surveillance system. To do so, we performed three different experiments with both systems in various practical situations. These experiments serve to answer the following questions:

- How do the conventional and TC systems scale as the number of targets increases?
- What are the benefits due to TC target prioritization, under conditions of system saturation?
- How effective is TC frame-cache affinity towards improving performance?

For the purposes of this evaluation, we use the metric (defined earlier in Section 3) *target throughput*. The target throughput is the number of target completion during an inter-frame interval, where target completion means execut-

ing all the trackers associated with a target. The target throughput percentage is the ratio of target completion to targets. Assuming the frame rate of a video camera stream is fixed and number of targets is constant, the target throughput will remain steady if the system is underloaded. This is because the system has enough resources to process all the targets during each inter-frame interval. However, when the system is overloaded, i.e., the system does not have enough resources to process all the targets during an inter-frame interval, some targets may not be fully executed (i.e., some trackers following a target may not have been executed) before the next frame arrives. In this case, the overall target throughput will decrease as new targets are added since each target handler will have less of a chance to be executed at each frame arrives.

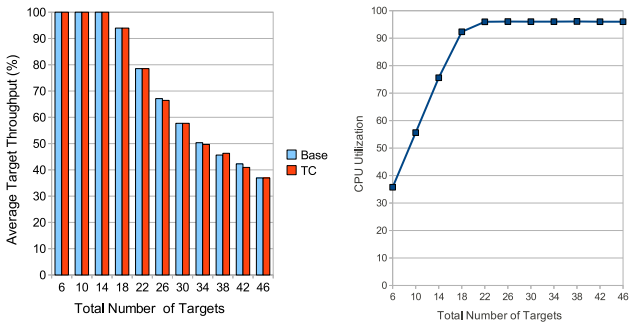
5.1 Experimental Setup

The experiments are conducted on Linux kernel 2.6.32 with an Intel Q6600 2.40GHz CPU with four cores. To reduce the noise in the experiments, we dedicated one core to run all the detector handlers for all the cameras, as well as the internal threads needed by the TC runtime system (e.g., to carry out merger operations). Incidentally, the TC runtime system uses a daemon thread to periodically invoke the equality checker handler provided by the application to determine if TCs have to be merged. The remaining three cores are dedicated for running the tracker handlers using the worker thread pool that we described in the previous section. To ensure that the first core is not overloaded, we emulate physical cameras by replaying video files with a fixed frame rate.

In our setup, a video camera emulator processes each video frame image to differentiate between foreground and background image. This is the first processing step in a video surveillance pipeline and would typically run on smart cameras under a real-world deployment scenario. In our setting, it takes about 150 ms to process foreground detection for one 800x600 image. A blob entrance detector (which would also typically run on smart cameras) identifies new objects in each video frame, and takes 20ms on an average. To safely provide a fixed video frame rate for our experiments, we set the frame rate of the video camera emulator to five frames per second. This rate provides enough time to process foreground detection and blob entrance detection, in our experimental testbed. As we mentioned earlier, we use a color tracker based on the *mean shift* algorithm [6] to track targets. The color tracker implementation from the OpenCV library takes 30-50ms to track a target in our setting. Assuming a 30x30 pixel object within a 800x600 pixel video, we set the overhead for tracking a target to be 30ms. To provide a fair comparison, these settings are fixed for all of our experiments.

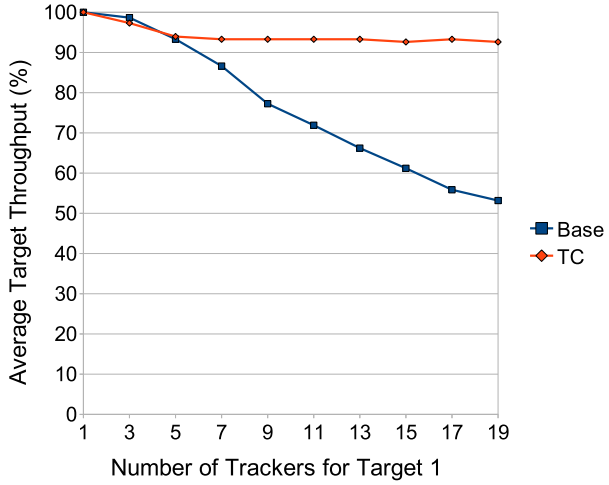
5.2 Target Throughput Scaling

Base Scenario. In this experiment, we measure the effects of increasing load on the surveillance system under test. Figure 7 shows the relationship of CPU utilization and average target throughput. Until the CPU utilization increases to over 90%, the system is underloaded and the average target throughput does not change even though new targets are added. However, as the number of targets increase from 14 to 18, the average target throughput starts to decrease due to system overload. Note that TC system and Baseline system[1] do not show any difference in this experiment since every target has only one tracker (i.e., is in the FOV of exactly one camera).



Left graph shows average throughput in terms of percent of target completion as number of targets is increased. Right graph reports CPU utilization results for same experiments. As system is overloaded, throughput decreases.

Figure 7: Effects of System Load on Target Handler Throughput



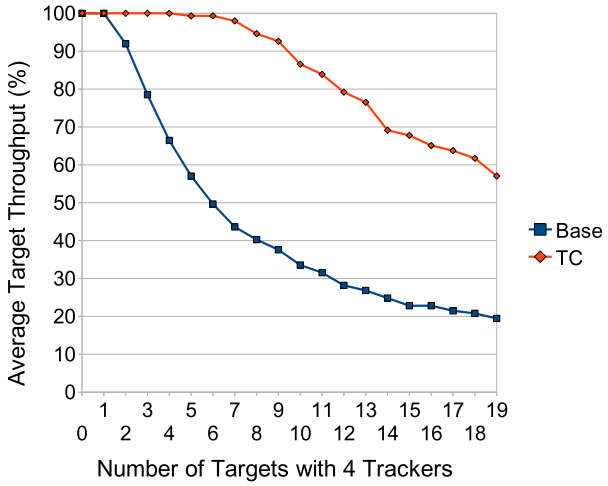
Average throughput in terms of percent target completion is reported as number of trackers for one of the targets increases. TC is target-aware and schedules targets evenly to maximize throughput, while single target consumes proportionally more resources in Base, degrading overall throughput.

Figure 8: Effects of Single Target in FOV of Multiple Cameras

Single Target in FOV of Multiple Cameras. In this experiment, we assume a situation where one target is in the FOV of multiple cameras while other targets are in the FOV of a single camera. To evaluate relative scalability between the TC and Baseline systems, we measure the average target throughput of TC and Baseline systems as the number of trackers for the first target increases. Note that the number of targets do not change while the number of trackers increases. Figure 8 shows the average target throughput with 14 targets for both TC and Baseline systems.

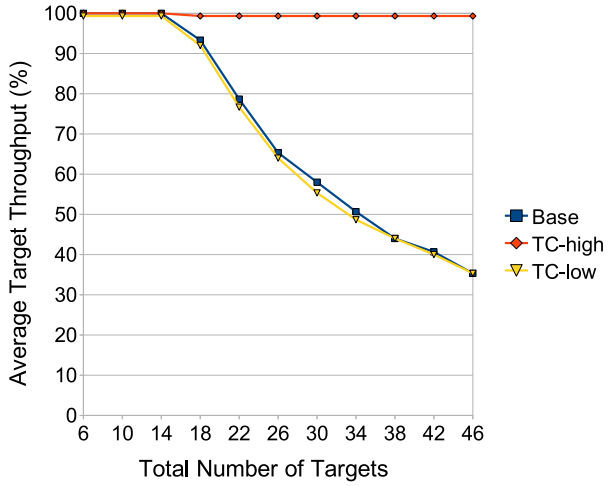
As depicted in Figure 7, both systems can run 14 trackers without throughput saturation. However, the average target throughput of the Baseline system starts to degrade

[1]We refer to the conventional thread-based implementation as the "Baseline" system.



Average throughput in terms of percent target completion is reported as number of targets with four trackers is increased. TC is target-aware and schedules targets evenly to maximize throughput, while single target consumes proportionally more resources in Base, degrading overall throughput. As shows in the figure, TC degrades more gracefully, than Base.

Figure 9: Effects of Many Resource Hungry Targets



Average throughput in terms of percent target completion is reported where one target is a high priority target (TC-High), and all others are low priority (TC-Low and Base). TC is target-aware and guarantees that the high-priority target is tracked even as the overall target handler throughput degrades under overload conditions. For low-priority targets, the TC system throughput degrades similar to Base.

Figure 10: Effects of Target Prioritization

as soon as additional trackers are added to the first target, since the system is overloaded beyond 14 trackers. However, the average target throughput of the TC system does not degrade much, because only the first target's throughput will be affected when more trackers are added to it. As shown in Figure 8, the target throughput of the Baseline system drops to nearly 50% of the original target throughput measured in an underloaded condition, because the number of trackers are increased. On the other hand, the average target throughput of the TC system remains well over 90% because the number of targets remains the same. This result demonstrates that the TC system fairly gives all targets of equal priority equitable amount of computational resources when the system is overloaded.

Many Resource Hungry Targets. The next experiment represents a situation where the number of resource hungry targets that are in the FOV of multiple cameras is increased. In this experiment, ten targets with one tracker (i.e., in the FOV of one camera) are initially running and we gradually increase the number of targets with four trackers (in the FOV of four cameras). Figure 9 shows the different scalability for both systems in terms of the average target throughput.

Initially, both the TC and baseline systems show 100% of average target throughput since the systems are underloaded. However, the baseline system starts to degrade once a target with four trackers is added, i.e. it hits the 14 tracker system saturation point. While the TC system also starts to degrade when more than four resource hungry targets are added, it degrades much more gracefully. For both systems, the average target throughput decreases once the point of saturation is reached, since both the number of targets and the number of trackers are increasing. However, since in this experiment the number of trackers increases faster than the number of targets, the TC system provides higher average target throughput than the baseline.

5.3 Target Prioritization

The next experiment assigns different priorities (high or low) to different targets. In this scenario, we assume there exists one target that is more important to track than the others. Regardless of system load state, the target tracking algorithm should successfully track the high-priority target. To emulate this scenario, we give high priority to one of the targets running on the system. The rest of targets have the same low priority. Although the TC system can allocate resources based on a gradient scale of target priorities, we only choose two priorities (high and low), for the purpose of simplicity. All targets have one tracker (i.e., in the FOV of a single camera) in this experiment. At initialization, six targets are being tracked under the TC and baseline systems. Figure 10 presents the results of this experiment.

When the experiment begins, all targets are subject to the same target throughput regardless of priority since both systems are not yet saturated. Beyond 14 targets, the systems reach saturation, and the low priority targets in the TC system experience reduced throughput. In the baseline system, all targets experience reduced throughput performance. However, the high priority target in the TC system still receives enough resources to be successfully tracked throughout the experiment. The low priority targets in the TC system have slightly lower average target throughput than the baseline system since the high priority target receives more resources than the others. However, only a few selective targets will likely require a higher level of attention in most practical situations and therefore, the loss in average target throughput for low priority targets is likely to be considered an acceptable trade-off.

5.4 Frame-Cache Affinity

Modern parallel machines are equipped with large caches to mitigate the significant access latency to main memory. Exploiting the caches can significantly benefit a video surveillance system. As we mentioned earlier, the TC system uses

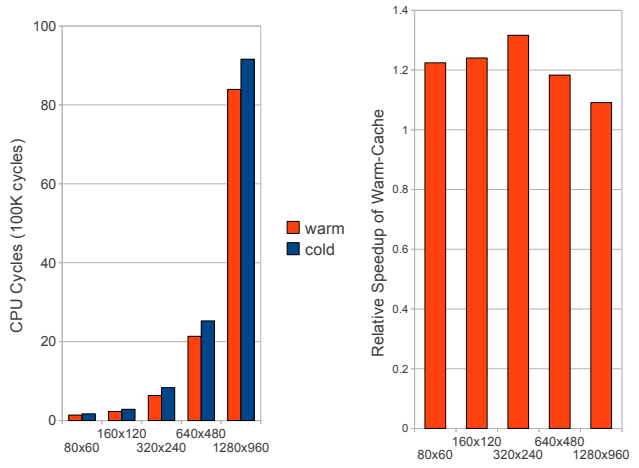


Figure 11: Effects of Cache Affinity Scheduling

cache affinity scheduling so long as target priorities are not violated. Essentially, the TC scheduler tries to schedule a tracker that can benefit from the cache being "warm" with the frame that it wants to process. Upon the completion of a tracker by a worker, the TC system tries to find another tracker that also needs to process the same frame and assigns it to the worker. The cache misses incurred by a tracker once the frame is pre-loaded is pretty insignificant. Since tracker code is compute-intensive, this "pre-loading" of the frame can be significant performance benefit.

To validate this design choice, we measured the CPU cycles for a tracker execution with various frame sizes on Intel Core I7 870 2.93GHz machine that runs Ubuntu 10.04 LTS (Linux kernel 2.6.32). We would like to show the number of cache load misses for different levels of the cache for a given video frame. Unfortunately, the latest version of PAPI [19] does not support L3 cache miss measurement. Moreover, the sizes of L1 and L2 cache are 32KB and 256KB, which are too small to show the benefit of frame cache affinity since each uncompressed color frame with 640x480 is about 1MB (640 x 480 x 3 bytes). For these reasons, we measured instead the CPU cycles for tracker execution in two different cases; *warm-cache* and *cold-cache*.

Figure 11 shows the results of tracker execution time (in CPU clock cycles) for different frame sizes for warm cache and cold cache. As can be seen from the figure, warm cache performs about 20% better than cold cache in most cases. This shows the potential benefit of cache-affinity scheduling with the TC system.

5.5 Summary of Results

Through the experimental study, we have shown that the TC system, being target-aware, provides better performance (measured in target throughput) than a conventional system for video surveillance in the presence of resource constraints. We have also shown that the priority framework of the TC system comes in handy to provide preferential target throughput for higher priority targets under resource constraints. We have also demonstrated the potential of the TC system to exploit cache affinity in scheduling the trackers.

6. RELATED WORK

Our work on TC is most closely related in spirit to the work by Banga, et al. on Resource Containers [3]. TC shares with Resource Containers the concept of providing a higher-level abstraction for resource management informed by the higher layers of the software stack. In the case of Resource Containers, the authors separate resource management from process isolation, aggregating the management of resources to groups of processes to improve the request processing throughput for systems such as web servers. TC utilizes target-awareness to inform resource management to improve the rate of target update processing under real-time constraints.

Other projects that are related to TC include ASAP [20]; the activity topology design based surveillance middleware [22]; and the high level abstractions for sensor network programming [15, 14]. ASAP [20] provides scalable resource management by using application-specific prioritization cues. Hengel et al. [22] approach scalability by partitioning the system according to an activity topology describing the observed (past) behavior of target objects in the network. EnviroSuite [15] and the work by Liu et al. [14] provide programming abstractions to shield the programmer from the chores of the distributed system (monitoring, leader selection, etc.) that are complementary the concerns in the TC system.

Through extensive research in the last two decades, the state of the art in automated visual surveillance has advanced quite a bit for many tasks including: detecting humans in a given scene [21, 7, 8, 23]; tracking targets within a given scene from a single or multiple cameras [10, 24]; following targets in a wide field of view given overlapping sensors; classification of targets into people, vehicles, animals, etc.; collecting biometric information such as face [26] and gait signatures [13, 25]; and understanding human motion and activities [9, 4, 16]. The TC programming model with its focus on enabling the prototyping of large-scale video-based surveillance applications complements these advances. We have been working with computer vision researchers in identifying the right level of abstractions in defining our programming model and are planning to open-source our development to make our programming system usable by the vision community.

7. CONCLUSION

In this paper, we have argued that building effective automated, large-scale video surveillance systems requires a new approach, with a focus on programmability and scalability. While considerable progress has been made in the area of computer vision algorithms, such advances have not translated to deployment in the large. We believe that this is due to the lack of adequate system abstractions and resource management techniques to ensure their performance. We believe *target containers* (TC) to be such an abstraction, and have presented it as the core contribution of this paper. Along with the TC abstraction, we have presented the TC API, programming model, and execution framework. We have demonstrated, through our TC prototype and experimental evaluation, the efficacy of our approach and the simplicity with which video surveillance applications can be constructed within the TC model. We have further demonstrated how the TC model allows the specification of target

tracking policies, such as target prioritization. Finally, our experimental results confirm the performance benefits of the TC approach.

8. REFERENCES

[1] OpenCV Wiki. `http://opencv.willowgarage.com/wiki/`, 2010.

[2] Bal, H. E., Kaashoek, M. F., and Tanenbaum, A. S. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18*, 3 (1992), 190–205.

[3] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: a new facility for resource management in server systems. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation* (Berkeley, CA, USA, 1999), USENIX Association, pp. 45–58.

[4] Bobick, A. F., and Davis, J. W. The recognition of human movement using temporal templates. *IEEE Trans. Pattern Anal. Mach. Intell. 23*, 3 (2001), 257–267.

[5] Choi, E. Lee, C. Feature extraction based on the bhattacharyya distance. *INTERNATIONAL GEOSCIENCE AND REMOTE SENSING SYMPOSIUM 5* (2000), 2146–2151.

[6] Comaniciu, D., Ramesh, V., and Meer, P. Real-time tracking of non-rigid objects using mean shift. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 2* (2000), 2142.

[7] Elgammal, A., Harwood, D., and Davis, L. Non-parametric model for background subtraction. In *FRAME-RATE Workshop, IEEE* (2000), pp. 751–767.

[8] Gavrila, D. Pedestrian detection from a moving vehicle. In *ECCV '00: Proceedings of the 6th European Conference on Computer Vision-Part II* (London, UK, 2000), Springer-Verlag, pp. 37–49.

[9] Gavrila, D. M. The visual analysis of human movement: a survey. *Comput. Vis. Image Underst. 73*, 1 (1999), 82–98.

[10] Haritaoglu, I., Harwood, D., and Davis, L. W4: Who? when? where? what? a real time system for detecting and tracking people. *Automatic Face and Gesture Recognition, IEEE International Conference on 0* (1998), 222.

[11] Johnson, K. L., Kaashoek, M. F., and Wallach, D. A. Crl: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), ACM, pp. 213–226.

[12] Kontothanassis, L., Stets, R., Hunt, G., Rencuzogullari, U., Altekar, G., Dwarkadas, S., and Scott, M. L. Shared memory computing on clusters with symmetric multiprocessors and system area networks. *ACM Transactions on Computer Systems 23*, 3 (2005).

[13] Little, J., and Boyd, J. E. Recognizing people by their gait: The shape of motion. *Videre 1* (1996), 1–32.

[14] Liu, J., Chu, M., Liu, J., Reich, J., and Zhao, F. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing 2*, 4 (2003), 50–62.

[15] Luo, L., Abdelzaher, T. F., He, T., and Stankovic, J. A. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Trans. Embed. Comput. Syst. 5*, 3 (2006), 543–576.

[16] Moeslund, T. B., and Granum, E. A survey of computer vision-based human motion capture. *Comput. Vis. Image Underst. 81*, 3 (2001), 231–268.

[17] Nikhil, R. S. Cid: A parallel, "shared memory" C for distributed-memory machines. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, 1995), Springer-Verlag, pp. 376–390.

[18] Nikhil, R. S., Ramachandran, U., Rehg, J. M., Halstead, Jr., R. H., Joerg, C. F., and Kontothanassis, L. I. Stampede: A programming system for emerging scalable interactive multimedia applications. In *LCPC '98: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing* (London, UK, 1999), Springer-Verlag, pp. 83–99.

[19] S., B., C., D., G., H., and P., M. Papi: A portable interface to hardware performance counters. *Proceedings of Department of Defense HPCMP Users Group Conference* (1999).

[20] Shin, J., Kumar, R., Mohapatra, D., Ramachandran, U., and Ammar, M. ASAP: A camera sensor network for situation awareness. In *OPODIS'07: Proceedings of 11th International Conference On Principles Of Distributed Systems* (2007).

[21] Stauffer, C., and Grimson, W. Adaptive background mixture models for real-time tracking. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 2* (1999), 2246.

[22] van den Hengel, A., Dick, A., and Hill, R. Activity topology estimation for large networks of cameras. In *AVSS '06: Proceedings of the IEEE International Conference on Video and Signal Based Surveillance* (Washington, DC, USA, 2006), IEEE Computer Society, p. 44.

[23] Viola, P., and Jones, M. Robust real-time object detection. In *International Journal of Computer Vision* (2001).

[24] Wren, C., Azarbayejani, A., Darrell, T., and Pentland, A. Pfinder: Real-time tracking of the human body. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19* (1997), 780–785.

[25] Zhang, R., Vogler, C., and Metaxas, D. Human gait recognition. *Computer Vision and Pattern Recognition Workshop 1* (2004), 18.

[26] Zhao, W., Chellappa, R., Phillips, P. J., and Rosenfeld, A. Face recognition: A literature survey. *ACM Comput. Surv. 35*, 4 (2003), 399–458.